VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Záznam, zpracování a zobrazení řídících parametrů auta

Recording and Processing of Car Control Parameters

2018 Pavel Ptáček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Pavel Ptáček**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Záznam, zpracování a zobrazení řídících parametrů auta
Recording and Processing of Car Control Parameters

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je navrhnout GUI aplikaci pro záznam a zobrazení zaznamenaných jízdních parametrů modelu auta. Aplikace bude v online i offline režimu umožňovat zobrazovat řídící údaje z jízdy vozu, včetně záznamu z řádkové kamery. Záznamy bude možno procházet v celé časové ose a bude možno porovnat více záznamů.

1. Vyberte vhodnou knihovnu pro implementaci GUI pro zobrazování záznamu jízdy auta.
2. Navrhněte celkovou koncepci programu, aby bylo možno pohodlně přidávat další zobrazované údaje.
3. Dle navržené koncepce implementujte aplikaci s vybranou grafickou knihovnou.
4. Vyzkoušejte aplikaci pro záznam dat pomocí více rozhraní.
5. Vyhodnoťte stabilitu, přehlednost, konfigurovatelnost aplikace a snadnost ovládání.

Seznam doporučené odborné literatury:

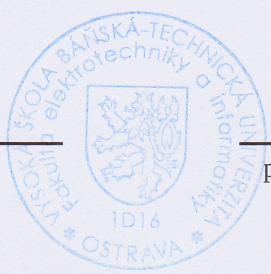
[1] Grafická knihovna QT, https://www.qt.io/
[2] Grafická knihovna GTK, https://www.gtk.org
[3] Matthew N., Stones R, Linux - Začínáme programovat, Computer Press, 2008, ISBN: 9788025119334

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Petr Olivka, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018



doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 26. dubna 2018 .....

Rád bych na tomto místě poděkoval mému vedoucímu práce Ing. Petru Olivkovi, Ph.D. za rady a velmi užitečnou pomoc při řešení problémů.

Abstrakt

Tato bakalářská práce se zabývá vytvořením aplikace s grafickým uživatelským rozhraním pro záznam, zpracování a zobrazení řídících parametrů auta. Práce se zabývá výběrem grafické knihovny, vytvořením grafického rozhraní, implementací a řešením problémů a na závěr testováním aplikace.

Klíčová slova: C++, Qt, GUI

Abstract

This bachelor work deals with creation of application with graphical user interface for recording, processing and displaying car control parametrs. Thesis deals with selection of graphical library, creation of graphical interface, implementation and problem solving and finally testing of application.

Key Words: C++, Qt, GUI

Obsah

Seznam použitých zkratek a symbolů **8**

Seznam obrázků **9**

Seznam výpisů zdrojového kódu **10**

1 Úvod **11**

2 Model auta **12**

3 Grafická knihovna **14**
3.1 OpenCV . . . 14
3.2 GTK+ . . . 14
3.3 Qt . . . 14
3.4 Výběr grafické knihovny . . . 14

4 Návrh koncepce programu **16**
4.1 Práce s aplikací . . . 16
4.2 Komponenty aplikace . . . 16
4.3 Logický náhled na aplikaci . . . 16

5 Implementace aplikace **19**
5.1 Implementované třídy . . . 19
5.2 Uspořádání grafických prvků . . . 19

6 Třídy, datové struktury a jejich popis **21**
6.1 Třída `Chart` a její potomci . . . 21
6.2 Třída `StaticRecord` a její potomci . . . 22
6.3 Třída `DataLoader` . . . 22
6.4 Popis sady řídících metod auta . . . 25
6.5 Popis `CarData.h` . . . 27

7 Datové toky **28**
7.1 Předávání dat . . . 28
7.2 Ukládání dat do souboru . . . 31

8 Grafické uživatelské rozhraní **33**
8.1 Hlavní okno . . . 34
8.2 Informační hlavička . . . 36
8.3 Statické záznamy . . . 36

8.4 Grafy 36

9 Problémy při implementaci a jejich řešení **38**
9.1 Výpočetní náročnost 38
9.2 Zpožděné pakety 38

10 Testování **41**
10.1 Testování s modelem auta 41
10.2 Testování výpadků paketů 41

11 Závěr **44**

Literatura **45**

Přílohy **45**

A Hlavičkový soubor `tfc.h` **46**

B CD **51**

Seznam použitých zkratek a symbolů

GUI	–	Graphical User Interface - Grafické uživatelské rozhraní
UDP	–	User Datagram Protocol - internetový protokol

Seznam obrázků

1 Model auta 13
2 Komponenty aplikace a datové toky mezi nimi 17
3 Třídní diagram 20
4 Okno programu s načtenými daty ze souboru 34
5 Menu File 35
6 Menu Load 35
7 Menu Window 35
8 První dráha 42
9 Druhá dráha 42

Seznam výpisů zdrojového kódu

1 Konstruktor třídy `ChartImg` a definice výchozí lambda funkce 21
2 Čtení dat ze souboru a jejich řazení 23
3 Struktura `tfc_protocol_empty_s` 25
4 Struktura `tfc_data_s` 25
5 Struktura `tfc_control_s` 26
6 Výčtový typ `tfc_andata_chnl_enum` 26
7 Hlavičkový soubor `CarData.h` 27
8 Definice signálů v hlavičkovém souboru `DataLoader.h` 28
9 Definice některých slotů v hlavičkovém souboru `MainWindow.h` 29
10 Příklad inicializace objektů grafů 30
11 Příklad použití ukazatele na vector s daty s ukazatelem do struktury `CarData` . . 30
12 Příklad tvorby prvků GUI 33
13 Výpis hlavičkového souboru `tfc.h` 46

1 Úvod

Cílem této bakalářské práce je navrhnout aplikaci s grafickým uživatelským prostředím pro záznam, zpracování a zobrazení zaznamenaných jízdních parametrů modelu auta.

Tato aplikace bude sloužit především těm, kteří se podílejí na programování autonomního řízení modelů aut a poté s nimi snaží uspět na soutěžích. Díky této aplikaci budou mít možnost analyzovat odesílaná data z modelu auta. Na základě této analýzy budou moci optimalizovat algoritmy pro řízení auta.

První část této práce je věnována popisu modelu auta, který byl použit pro testování aplikace. Díky jízdě, zaznamenání jízdních parametrů a poskytnutí těchto dat byl vývoj a testování aplikace jednodušší.

Druhá část se zabývá výběrem grafické knihovny, která byla použita pro vývoj aplikace. Rozhodnutí probíhalo mezi dvěmi nejznámějšími knihovnami. Nejprve budou jednotlivé knihovny mezi sebou porovnány a poté vysvětleno samotné rozhodnutí výběru.

Ve třetí části práce je popsána koncepce programu. Dále je čtenář podrobně seznámen s jednotlivými třídami, hlavičkovými soubory, strukturami a také metodami, které se v aplikaci používají pro komunikaci mezi objekty a manipulaci s daty.

Čtvrtá část je věnována popisu grafického uživatelského rozhraní a popisu funkcí jednotlivých ovládacích prvků.

V páté části jsou popsány různé problémy, které nastaly při implementaci nebo testování. U těchto problémů jsou poté podrobně vysvětleny možné postupy řešení.

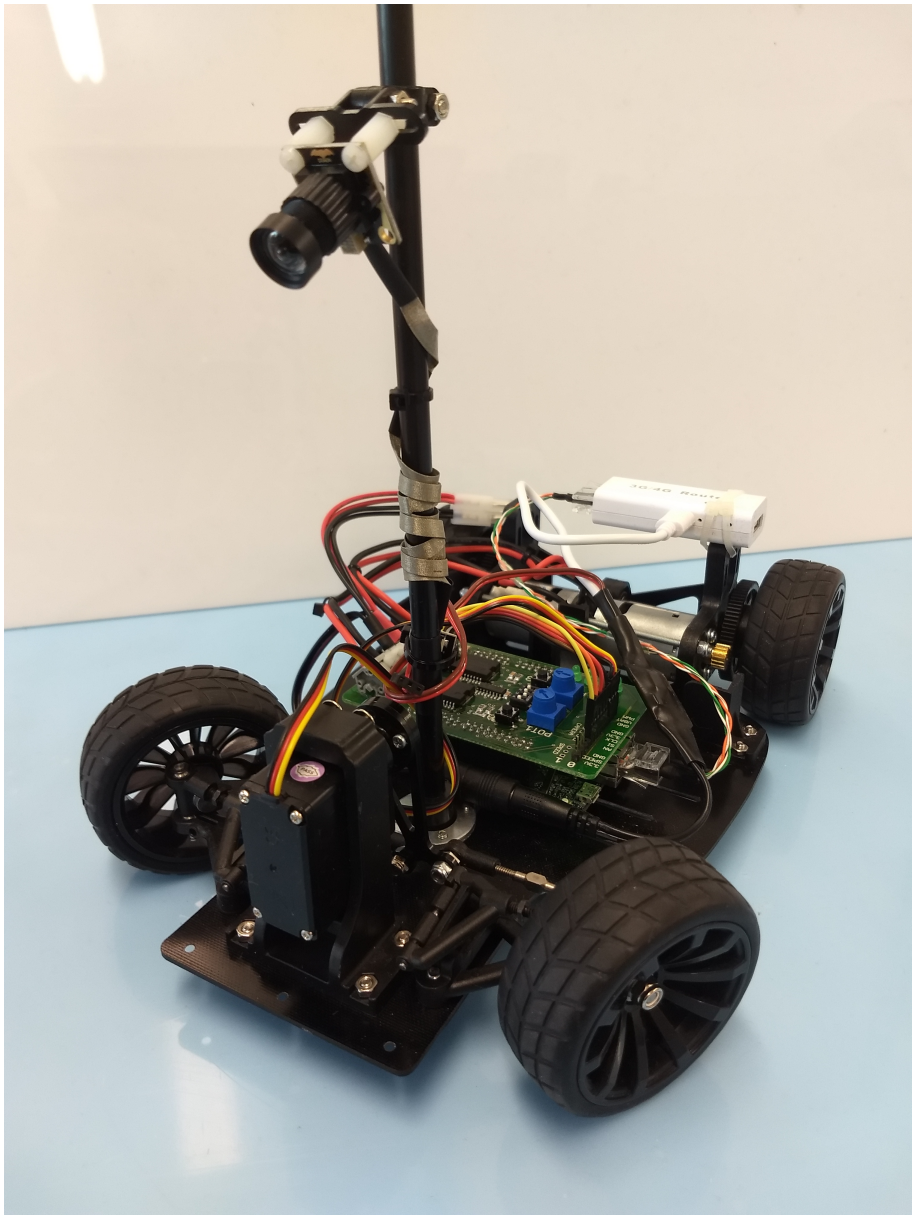
Poslední část se věnuje testování aplikace a porovnání algoritmů, které byly v průběhu implementace vymyšleny.

V závěru této práce shrnu funkce, které byly v aplikaci implementovány. Dále vyhodnotím jak je aplikace stabilní a zda je přehledná. Rovněž se zaměřím na zhodnocení konfigurovatelnosti aplikace, především přidávání dalších zobrazovaných údajů. Jako poslední zhodnotím jednoduchost a efektivitu ovládání.

2 Model auta

Pro testování aplikace popisované v této práci byl použit model automobilu, který je možno vidět na obrázku 1. Model automobilu je 30,5 cm vysoký, 17,5 cm široký a na délku má 24 cm. Obsahuje základní desku s programovatelným čipem, na kterém je nahrán program ovládající auto. Dva elektrické motory pohánějí zadní kola a přední kola jsou ovládány servomotorem. Energii dodává baterie umístěna v zadní části auta.

Auto posílá v pravidelném intervalu dvě datové struktury, které obsahují různé řídící informace. Tyto informace se v aplikaci dále zpracují a následně zobrazí. Používané struktury jsou dále podrobně popsány v kapitole 6.4.



Obrázek 1: Model auta

3 Grafická knihovna

Prvním ze zadaných úkolů byl výběr grafické knihovny pro implementaci GUI. Množství grafických knihoven pro vývoj v programovacím jazce C++ je poměrně velké. Mezi nejznámější a nejpoužívanější knihovny patří OpenCV, GTK+ a Qt. Tyto tři knihovny nyní popíšu.

3.1 OpenCV

OpenCV [3] je grafická knihovna primárně zaměřena na zpracování obrazu v reálném čase. Tato knihovna je vyvíjena v programovacím jazyce C++ a je šířena pod BSD licencí. Jedná se o multiplatformní knihovnu, která podporuje velké množství jak desktopových, tak mobilních operačních systémů.

3.2 GTK+

GTK+ [2] knihovna původně vznikla pro potřeby grafického editoru GIMP. Později byla použita pro prostředí GNOME a díky tomu se stala jednou z nejznámějších knihoven. Knihovna je vyvíjena v programovacím jazyce C a je multiplatformní, podporuje GNU/Linux, Unixové systémy, Windows a macOS. Také je lokalizována do mnoha jazyků včetně češtiny. Knihovna je šířena jako open source s licencí LGPL.

3.3 Qt

Qt knihovna [1] je vyvíjena v programovacím jazyce C++, ale pro vývoj aplikací v této knihovně je podporováno velké množství jazyků jiných, např. Pyhon, Ruby, C, C#, Java, Haskell. Jedná se o multiplatformní knihovnu, která navíc oproti GTK+ podporuje i mobilní operační systémy jako Android, iOS nebo Windows Phone. Možnosti licencování jsou rozsáhlejší než u GTK+, jelikož aplikace je možno distribuovat pod licencí GPL, LGPL nebo komerčně. Qt ovšem není pouze grafická knihovna, ale kompletní aplikační framework, který nabízí velké množství nástrojů pro práci se soubory, sítí, grafikou, multimédií, vlákny, SQL, či zpracování XML. Velkou výhodou Qt oproti GTK+ je její dokumentace, která je zpracována kompletně a velmi přehledně. Pro vývoj je možné použít oficiální vývojové programy QtCreator a QtDesigner.

V této knihovně je vyvíjena spousta aplikací, od malých projektů po projekty velké, např. Skype, Google Earth, Linuxové prostředí KDE, webový prohlížeč Opera, virtualizační software VirtualBox a další.

3.4 Výběr grafické knihovny

Před vývojem aplikace, která je cílem této práce, se pro záznam a zobrazení jízdních parametrů auta používala aplikace vyvíjená v OpenCV. Vzhledem k tomu, že tato knihovna nenabízí možnosti pro tvorbu rozsáhlého uživatelského rozhraní, tak musela být vybrána knihovna jiná. Po zvážení všech kladů a záporů zbývajících dvou knihoven jsem se rozhodl tuto práci vyvíjet

pomocí grafické knihovny Qt, především z důvodu kvalitní dokumentace, oficiálního vývojového prostředí QtCreator a několika nástrojů, které usnadňují práci se soubory, sítí a grafikou.

4 Návrh koncepce programu

4.1 Práce s aplikací

Po zapnutí aplikace se zobrazí hlavní okno, kde v horní části je umístěno hlavní menu. Menu bude obsahovat položky pro spuštění načítání dat, otevření nového okna, uložení načteného záznamu do souboru a zavření aktuálního okna.

Pro spuštění příjímání a zobrazování dat si uživatel v menu hlavního okna aplikace vybere zdroj, ze kterého požaduje získávat data. Na výběr budou tři zdroje – soubor, síť, sériová linka. Po zvolení požadovaného zdroje se zobrazí dialog pro upřesnění parametrů. V případě zvolení čtení ze souboru se zobrazí dialog pro výběr souboru s daty. V případě volby načítání ze sítě nebo sériové linky se zobrazí dialog pro specifikování síťového nebo sériového portu. Po zadání správných parametrů se začnou načítat data. Po načtení určitého bloku dat se tyto data zobrazí v hlavním okně aplikace. V hlavním okně se zobrazí záznam z řádkové kamery a ostatní číselná data se vykreslí ve formě grafů. Pokud uživatel vybere nějaký řádek záznamu v grafu, zobrazí se nad grafy detailní informace o tomto záznamu. Nad každým grafem bude vypsána hodnota parametru, který daný graf vykresluje.

Pro uložení načteného záznamu uživatel v menu hlavního okna vybere možnost uložení do souboru. Zobrazí se dialog, ve kterém specifikuje adresář a výstupní soubor. Uživatel bude mít na výběr dva způsoby uložení – uložení celého záznamu nebo pouze části záznamu. Pro uložení části záznamu uživatel tuto oblast označí. Pokud žádná část označená nebude, uloží se celý záznam.

Pro porovnání několika záznamů uživatel vybere v menu aplikace otevření nového okna. V tomto novém okně bude mít uživatel možnost načíst jiný záznam a tento záznam poté porovnávat s předchozím.

4.2 Komponenty aplikace

Aplikace bude rozdělena do několika komponent, které budou mezi sebou komunikovat a předávat si data.

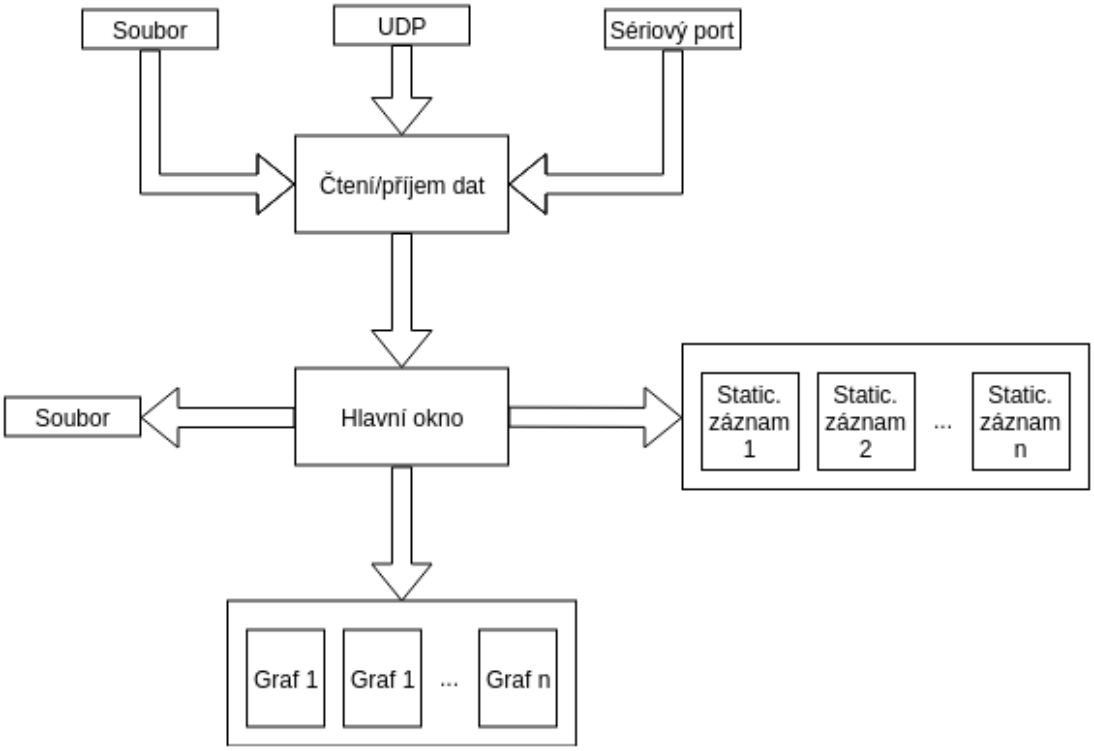
Jednotlivé komponenty a tok dat mezi nimi jsou znázorněny na obrázku 2.

4.3 Logický náhled na aplikaci

Jednotlivé komponenty budou rozděleny do několika tříd. Každá třída bude mít za úkol provádění nějakého úkolu. Třídy budou mezi sebou komunikovat a předávat si data.

Budou naimplementovány třídy zajišťující následující úkoly:

- Vykreslení uživatelského prostředí
- Načítání a příjem dat



Obrázek 2: Komponenty aplikace a datové toky mezi nimi

- Zobrazení dat – vykreslení grafů a záznamu z kamery
- Zobrazení informací o aktuálně vybraném záznamu

Třída, zajišťující vykreslení uživatelského prostředí, bude mít také na starost obsloužení uživatelských požadavků, např. kliknutí myši, stisknutí klávesy. Tato třída bude rovněž centrálním místem pro data. Veškerá data, která načte třída pro načítání dat, budou poslány této třídě.

Ve třídě pro čtení a příjem dat bude implementováno čtení dat ze tří zdrojů – soubor, síť, sériová linka. Pro každý z těchto zdrojů bude ve třídě metoda, která po zavolání spustí dané načítání. Po načtení určitého bloku dat se tyto data pošlou hlavní třídě, ve které se data shromažďují. Ve třídě bude také naimplementováno zastavení čtení dat ze sítě nebo ze sériové linky.

Zobrazování dat bude probíhat formou vykreslování grafů. Každý graf bude vykreslovat hodnoty jednoho řídícího parametru auta. Hodnoty parametrů mohou být více druhů – záznam z kamery, znaménkové hodnoty a bezznaménkové hodnoty. Z tohoto důvodu bude nutné definovat více tříd, kdy každá třída bude mít implementováno vykreslování jednoho z těchto druhů hodnot. Tyto třídy budou mít společné některé vlastnosti a metody, tudíž bude nutné definovat bázovou třídu, ze které poté budou dědit třídy určené pro různé typy grafů. Mezi společné vlastnosti bude patřit například šířka grafu, barva grafu či minimální a maximální hodnoty, kterých mohou hodnoty vykreslovaného parametru nabývat. Společnými metodami bude například metoda pro spuštění vykreslování grafu či vykreslení označeného řádku.

Pro zobrazování informací o aktuálně vybraném záznamu bude také implementováno více tříd. Rovněž bude definována bázová třída obsahující společné vlastnosti a metody, ale potomci této třídy budou pouze dva. První bude vykreslovat jeden řádek obrazu z kamery a druhý bude vypisovat číselnou hodnotu.

5 Implementace aplikace

V této kapitole budou stručně popsány třídy, které byly definovány při návrhu koncepce aplikace popsané v předchozí kapitole. Dále budou představeny některé Qt třídy a bude popsáno jejich použití s již existujícími třídami.

5.1 Implementované třídy

Jednou z nejdůležitějších tříd je třída `MainWindow`, která reprezentuje hlavní okno aplikace. Tato třída se stará o veškeré dění v programu. Jedná se například o vykreslení uživatelského prostředí nebo obsloužení uživatelských požadavků, jako je např. kliknutí myší nebo stisknutí kláves. Dále zajišťuje inicializaci různých objektů nebo předávání zpráv a dat mezi objekty.

Jako další je v programu třída `Chart` a její potomci. Tyto třídy se starají o vykreslování grafů a budou podrobně popsány v sekci 6.1.

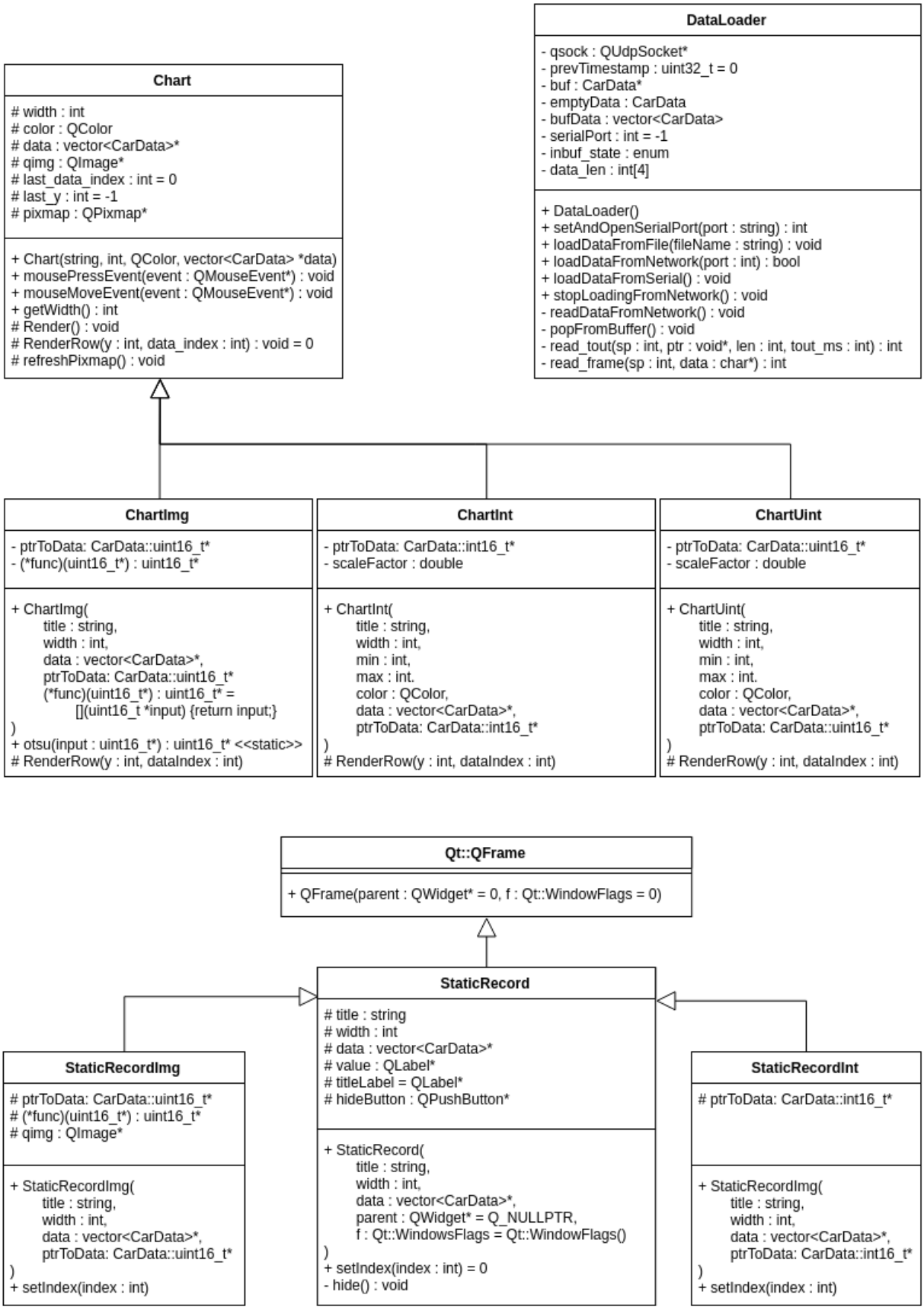
Další neméně důležitou třídou je třída `StaticRecord` a následně její potomci. Tyto mají za úkol zobrazovat informace o aktuálně vybraném záznamu. Podrobný popis je uveden v sekci 6.2.

Načítání a příjímání dat obstará třída `DataLoader`, ve které je naimplementováno čtení dat jak ze souboru, tak ze sítě pomocí UDP protokolu a také ze sériové linky. Tuto třídu rovněž podrobněji popíšu, a to v sekci 6.3.

Všechny zde popsané třídy, jejich metody a proměnné jsou zakresleny v třídním diagramu na obrázku 3.

5.2 Uspořádání grafických prvků

Aby se zajistilo pohodlné přidávání dalších zobrazovaných údajů, bylo nutné použít nějaký mechanismus, který se bude automaticky starat o uspořádání jednotlivých grafů a statických informací. O to se velmi dobře postarají třídy Qt knihovny – layouty. Díky použití Qt layoutů je přidání nebo odebrání zobrazovaných dat velmi pohodlné. V programu se používají dva typy layoutů: horizontální a mřížkový. Ve zdrojovém kódu hlavního okna jsou definovány tři hlavní layouty. Ve všech třech případech se jedná o stejný typ layoutu, a sice horizontální. Horizontální layout rozmisťuje své vnitřní prvky rovnoměrně po horizontální ose. Tento způsob je využíván u sekce s informační hlavičkou, statickými záznamy i grafy. Tyto sekce budou podrobněji popsány v kapitolách 8.2, 8.3 a 8.4. Druhý typ layoutu, který je v programu používán, je mřížkový. Ten se používá uvnitř třídy `StaticRecord`, která bude podrobněji rozepsána v kapitole 6.2. Po inicializaci objektů, které se mají rovnoměrně rozmísťovat, se jednoduše vloží do daného layoutu. Prvky jednotlivých layoutů se dle nastavení rodičovského layoutu samy zobrazí a rozmístí, tudíž se nemusí řešit vykreslování či počítání různých souřadnic. Díky těmto layoutům se také nemusí řešit změny velikosti okna. Se správným nastavením se o všechny tyto problémy stará Qt layout systém.



Obrázek 3: Třídní diagram

6 Třídy, datové struktury a jejich popis

V této kapitole budou podrobně popsány implementované třídy a použité datové struktury

6.1 Třída `Chart` a její potomci

Tato třída reprezentuje graf zobrazující jeden typ hodnot struktury `CarData`.

6.1.1 Popis

Jedná se o abstraktní třídu, ze které dědí třídy pro konkrétní typy grafů. Třída je potomkem Qt třídy `QLabel`, která je určena pro zobrazování textu nebo obrázku. V této třídě je definovaný konstruktor, který přijímá parametry pro nastavení třídních proměnných. Jedná se o parametry titulek, šířka grafu a ukazatel na vector s daty. Konstruktory potomků jsou doplněny o další položky. Mezi důležité metody třídy patří metody `refresh()`, `render()` a `renderRow(int y, int data_index)`. První zmíněná metoda je volána z hlavního okna aplikace, kdykoliv je potřeba aktualizovat data v grafu. Tato metoda zavolá metodu `render()`, která dokreslí nová data do obrázku datového typu `QImage`, a následně zavolá metodu, která se postará o zobrazení aktualizovaného obrázku. Uvnitř metody `render()` se provedou výpočty souřadnic, na které se mají nová data vykreslit a poté se pro každý záznam nových dat zavolá abstraktní metoda `renderRow(int y, int data_index)`, která je implementována v potomcích této třídy.

6.1.2 Třída `ChartImg` a předání ukazatele na funkci

Z auta přichází data z řádkové kamery, podle které se auto rozhoduje. Tyto data jsou zobrazena pomocí třídy `ChartImg`. Občas může být užitečné kromě obrazu získaného z kamery zobrazovat i nějaký zpracovaný obraz. Aby se všal nemusely pro každý druh zpracovaného obrazu psát další třídy, může se využít stávající třída `ChartImg`, která při vykreslování řádků použije na data nějakou funkci, která výsledná data upraví. V konstruktoru třídy `ChartImg` je navíc jeden volitelný parametr. Tento parametr je ukazatel na funkci, která se na data použije. Pokud tento parametr není specifikovaný, tak se použije výchozí lambda funkce. Tato lambda funkce nedělá s daty žádnou změnu, pouze vstupní data vrátí zpět. Definici konstruktoru a lambda funkce je možno vidět ve výpisu kódu 1. Ve třídě `ChartImg` je poté definována statická funkce `uint16_t* otsu(uint16_t *input)`, která obraz zpracuje. Při inicializaci grafu s filtrovaným obrazem kamery se tedy v konstruktoru předá reference na tuto statickou funkci. Při inicializaci grafu s nefiltrovaným obrazem se tento parametr vynechá. Pokud by bylo žádoucí na obraz použít jinou funkci, může se tato funkce definovat jako další statická funkce ve třídě `ChartImg` a poté se na ni předá reference, nebo se může tato funkce definovat jako lambda funkce přímo při předávání parametrů konstruktoru. Příklad inicializace `ChartImg` je možno vidět ve výpisu kódu 10.

```
ChartImg(
```

```
    string title,
    int width,
    vector<CarData> *data,
    uint16_t CarData::*ptrToData,
    uint16_t *(*func)(uint16_t*) = [](uint16_t *input) {return input;}
);
```

Výpis 1: Konstruktor třídy `ChartImg` a definice výchozí lambda funkce

6.2 Třída `StaticRecord` a její potomci

Tato třída reprezentuje statickou hlavičku zobrazující informace o aktuálně vybraném záznamu v grafu. Jedná se o potomka Qt třídy `QFrame`. `QFrame` jsem zvolil především kvůli rámečkového vzhledu, který se u tohoto objektu dá nastavit. `StaticRecord` je opět abstraktní třída, ze které dědí třídy, které zobrazují hodnoty pro konkrétní typy grafů.

Jelikož se jedná o potomka třídy `QFrame`, tak můžeme na tento widget aplikovat nějaký Qt layout a umístit do něj několik grafických prvků. V tomto případě jsem zvolil mřížkový layout `QGridLayout`. Uvnitř layoutu je umístěno tlačítko pro skrytí/odkrytí této hlavičky, titulek s názvem dat, která tato hlavička zobrazuje a také samotná data. Tyto data se mění na základě toho o jakého potomka se jedná. Jsou zde dva typy: `StaticRecordImg` a `StaticRecordInt`. Na rozdíl od třídy `Chart` jsou zde pouze dvě konkretizace, protože v případě, kdy se zobrazuje pouze jedna číselná hodnota, je zbytečné implementovat třídy jak pro známenkové, tak pro neznaménkové celé číslo.

6.3 Třída `DataLoader`

Jak již název napovídá, třída `DataLoader` má na starost veškeré načítání dat z různých zdrojů. Jak již bylo v předchozích kapitolách zmíněno, ve třídě je naimplementováno čtení dat ze tří zdrojů – ze souboru, ze sítě a ze sériové linky.

6.3.1 Načítání ze souboru

O čtení dat ze souboru se stará metoda `loadDataFromFile(string fileName)`. Tento typ načítání dat je na implementaci nejjednodušší. Při čtení ale musíme vzít v úvahu jeden možný problém. Záznamy v souboru mohly být uloženy při čtení dat ze sítě a jak se později v kapitole 9.2 dozvíme, některé záznamy mohou přicházet opožděně. Pokud se s těmito daty poté neprovedl žádný proces, tak mohou být v souboru uloženy neuspořádaně. Musí se tedy zajistit správné uspořádání těcho záznamů. To se provede setřízením záznamů podle položky `timestamp` ve struktuře `tfc_data_s`. Hodnoty této položky na sebe musí navazovat.

V metodě zmíněné v předchozím odstavci je deklarována lokální proměnná `result` typu `vector<CarData>`. Do tohoto vectoru se budou postupně ukládat načtené záznamy. Pro otevření

souboru a následné čtení používám příkazy `open` a `read`. Po úspěšném otevření souboru v režimu pouze pro čtení se ve smyčce začnou načítat data. Při každém průchodu smyčkou se přečte blok dat o velikosti jedné struktury `CarData`. Tento blok se uloží do již dříve deklarované struktury typu `CarData`. Nyní přichází na řadu kontrola souvislosti dat. Pokud ve vectoru `result` není žádný záznam nebo pokud je časová známka aktuálně čtených dat větší než časová známka posledního záznamu ve vectoru, tak se tento záznam vloží do vectoru na konec. Pokud by tato podmínka vyšla negativně, tak to znamená, že aktuální záznam nepatří na konec dat a musí se pro něj najít místo. Postupně se prochází vector `result` od konce do začátku a hledá se záznam, který má časovou známku menší než aktuálně čtený záznam. Po nalezení se vloží aktuálně čtený záznam za tento nalezený záznam. Ukázku kódu vykonávaného ve smyčce je možno vidět ve výpisu kódu 2.

```
while (read(sp, &data, sizeof(data)) > 0)
{
  if (result.size() == 0 ||
     data.structData.timestamp > result.back().structData.timestamp)
      result.push_back(data);
  else
  {
    for (vector<CarData>::iterator it = result.end() - 1; it != result.begin();
         it--)
      if (it->structData.timestamp < data.structData.timestamp)
      {
        it++;
        result.insert(it, data);
        break;
      }
  }
}
```

Výpis 2: Čtení dat ze souboru a jejich řazení

Po přečtení celého souboru a vystoupení ze smyčky jsou všechny záznamy ve vectoru `result` uspořádané, ale chybí v nich prázdné záznamy indikující výpadek. Z toho důvodu se vector se záznamy musí projít znovu od začátku do konce a pro každý záznam se zkontroluje, zda navazuje na předchozí, tedy zda je časová známka aktuálně kontrolovaného záznamu o jedno číslo větší než časová známka záznamu předchozího. Pokud tomu tak není, tak se před tento záznam vloží prázdný záznam. Po projití celého vectoru jsou v něm uloženy všechny záznamy ze souboru a také označeny všechny chybějící záznamy. Nyní se mohou tyto data poslat instanci hlavního okna, aby je bylo možnost zobrazit v grafech. Provede se to vysláním signálu `loaded(result)`. Použití signálů bude podrobněji vysvětleno v kapitole 7.1.1.

6.3.2 Načítání ze sítě

Načítání dat ze sítě probíhá ve více metodách. Uživatel používající tuto třídu volá veřejnou metodu `loadDataFromNetwork(int port)`, která se postará o zbytek. V parametru této metody se předá číslo portu, na kterém se má naslouchat. Uvnitř metody proběhne napojení na tento port a spustí se čtení dat ze sítě. Pro manipulaci se síťovým připojením používám Qt třídu `QUdpSocket`. Ve třídě je deklarována třídní proměnná `qsock` typu `QUdpSocket`. Pro napojení na port se používá metoda tohoto objektu `bind`, které se v parametrech předá IP adresa a port. Jelikož je zapotřebí naslouchat na všech síťových rozhraních, první parametr musí být `QHostAddress::Any`. Druhým parametrem je číslo portu přijaté v parametru metody `loadDataFromNetwork(int port)`. Samotné spuštění čtení dat se realizuje propojením signálu `readyRead()` objektu `qsock` a privátního slotu `readDataFromNetwork()` v této třídě `DataLoader`. Jakmile jsou na jakémkoliv rozhraní na dříve specifikovaném portu dostupná nějaká data, zavolá se metoda `readDataFromNetwork()`, která se postará o čtení a manipulaci s přijatými daty.

Vzhledem k tomu, že i u tohoto způsobu čtení dat je pravděpodobné, že přijatá data budou zpřeházená, je potřeba data speciálně zpracovat a uspořádat. To se provede v již dříve zmíněné metodě `readDataFromNetwork()` a `popFromBuffer()`. Tyto metody jsou podrobně rozepsány v kapitole 9.2.

6.3.3 Načítání ze sériové linky

Načítání dat ze sériové linky opět probíhá ve více metodách. Uživatel používá pouze jednu veřejnou metodu, a tou je `loadDataFromSerial(string port)`, kde v parametru předá port sériové linky. Pro samotné načítání používám opět příkazy `open` a `read`. Ve veřejné metodě se provede pokus o otevření sériového portu. Při úspěchu se začnou načítat data. Vzhledem k tomu, že čtení ze sériové linky je blokující, musí se vše provádět v samostatném vlákně. Pokud by se to neudělalo, tak by nebyla možná aktualizace grafů či interakce s uživatelským rozhraním do doby, než by načítání dat skončilo. To je samozřejmě nežádoucí, protože je potřeba přijímaná data vidět ihned. Pro práci s vlákny se mi jevilo jako nejvhodnější použít Qt třídu `QThread`. V metodě `loadDataFromSerial(string port)` se vytvoří nový objekt `thread` tohoto type. Aktuální objekt třídy `DataLoader` se do tohoto vlákna přesune pomocí třídní metody `moveToThread(thread)`, kterou zdědil od předka `QObject`. Jako další krok se musí propojit signál `started()` objektu `thread` se slotem `readDataFromSerial()` aktuální třídy. Tím se zajistí, že po spuštění vlákna se spustí metoda, která se postará o načítání dat ze sériového portu. Po propojení dalších signálů a slotů, které jsou pro vlákno důležité, se může toto vlákno spustit metodou `start()`.

6.4 Popis sady řídících metod auta

Data posílána z auta používají struktury definované v hlavičkovém souboru `tfc.h`. Tento soubor je přiložen v příloze A. V této kapitole tyto struktury a jejich položky popíšu.

Hlavičkový soubor obsahuje definici čtyř struktur: `tfc_protocol_empty_s`, `tfc_data_s`, `tfc_setting_s` a `tfc_control_s`. Struktura `tfc_setting_s` slouží k počátečnímu nastavení parametrů auta. V této práci se však nepoužívá, tudíž ji nebudu dále popisovat.

6.4.1 Struktura `tfc_protocol_empty_s`

Struktura `tfc_protocol_empty_s` se používá pro odesílání dat, kde se na místo položky `data` umístí jedna z ostatních zmíněných struktur. Definici struktury je možno vidět ve výpisu kódu 3.

Položky `stx` a `etx` jsou speciální bajty, určující start a konec paketu. Každý paket zde musí mít hodnoty konstant `::STX` pro `stx` a `::ETX` pro `etx`. Položka `length` informuje o délce celého paketu. Položka `cmd` určuje typ struktury, které je v tomto paketu posílána. Tato struktura je vložena na místo položky `data`.

```
struct tfc_control_s
{
   uint8_t stx;
   uint16_t length;
   uint8_t cmd;
   char data[0];
   uint8_t etx;
}
```

Výpis 3: Struktura `tfc_protocol_empty_s`

6.4.2 Struktura `tfc_data_s`

Struktura `tfc_data_s` je definována ve výpisu kódu 4.

Položka `timestamp` je časová známka struktury. Touto položkou se při příjmu dat zjišťuje, zda na sebe pakety navazují a zda nedošlo k výpadku. Položka `adc` je pole, které obsahuje analogová data specifikovaná ve výčtovém typu `tfc_andata_chnl_enum`, který je popsán v kapitole 6.4.4. Položka `dip_sw` sděluje hodnoty DIP přepínačů a položka `push_sw` hodnoty tlačítek. Důležitou položkou je položka `image`, která obsahuje obraz z řádkové kamery. Jedná se o jeden řádek o šířce `TFC_CAMERA_LINE_LENGTH`. Poslední položka `_padding` obsahuje pouze výplňové bajty.

```
struct tfc_data_s
{
    uint32_t timestamp;
```

```
    uint16_t adc[ anLast ];
    uint8_t dip_sw;
    uint8_t push_sw;
    uint16_t image[ TFC_CAMERA_LINE_LENGTH ];
    uint32_t _padding;
};
```

Výpis 4: Struktura `tfc_data_s`

6.4.3 Struktura `tfc_control_s`

Definice struktury `tfc_control_s` je vypsána ve výpisu kódu 5.

První položka v této struktuře je položka `leds`, která obsahuje hodnoty diod. Hodnoty položek `pwm_onoff` a `servo_onoff` mohou nabývat 0 nebo 1, kde 0 znamená vypnuto a 1 zapnuto. Tyto hodnoty indikují stav motorů a stav serv. Následuje položka `_padding1` obsahující jeden výplňový bajt. Předposlední položka `pwm` je pole obsahující hodnoty obou motorů. Poslední položka `servo_pos` je rovněž pole a obsahuje hodnoty indikující pozici serv.

```
struct tfc_control_s
{
    uint8_t leds;
    uint8_t pwm_onoff;
    uint8_t servo_onoff;
    uint8_t _padding1;
    int16_t pwm[ 2 ];
    int16_t servo_pos[ 2 ];
};
```

Výpis 5: Struktura `tfc_control_s`

6.4.4 Výčtový typ `tfc_andata_chnl_enum`

Výčtový typ `tfc_andata_chnl_enum` obsahuje šest položek a jeho definici je možno vidět ve výpisu kódu 6.

Položky v tomto výčtovém typu definují pořadí analogových dat v poli `adc` ve struktuře `tfc_data_s`. Položky `anPOT_1` a `anPOT_2` značí první a druhý potenciometr. Položky `anFB_A` a `anFB_B` značí zpětné vazby. Položka `anBAT` značí napětí baterie a poslední položka `anLast` se používá pouze jako velikost pole.

```
enum tfc_andata_chnl_enum
{
    anPOT_1,
```

```
    anPOT_2,
    anFB_A,
    anFB_B,
    anBAT,
    anLast
};
```

Výpis 6: Výčtový typ `tfc_andata_chnl_enum`

6.5 Popis `CarData.h`

Struktura `CarData` definovaná v hlavičkovém souboru `CarData.h` se používá pro pohodlný přístup ke všem datům ze záznamu. Tato struktura spojuje struktury `tfc_data_s` a `tfc_control_s` do jedné. Definice této struktury je zobrazena ve výpisu kódu 7.

```
#include "tfc.h"

struct CarData
{
    uint8_t stx1;
    uint16_t length1;
    uint8_t cmd1;
    tfc_data_s structData;
    uint8_t etx1;
    uint8_t stx2;
    uint16_t length2;
    uint8_t cmd2;
    tfc_control_s structControl;
    uint8_t etx2;
};
```

Výpis 7: Hlavičkový soubor `CarData.h`

7 Datové toky

7.1 Předávání dat

V aplikaci se musí často předávat různá data mezi různými objekty. Předávání dat se musí provádět při přijetí nových dat, při aktualizaci dat v grafech a při zobrazování informací o právě zvoleném záznamu. Tyto data se shromažďují na jednom místě, z kterého se následně předávají dál. Toto centrální místo je instance objektu reprezentující hlavní okno – `MainWindow`. Tato třída má třídní proměnnou typu `vector`, jehož prvky jsou struktury `CarData`.

7.1.1 Použití Qt signálů a slotů

Qt signály a sloty se používají pro komunikaci mezi různými objekty, případně i v rámci jednoho objektu. Tento mechanismus se používá následovně. Třída, která bude signál vysílat, musí mít definovanou třídní metodu typu `Qt::SIGNAL`. Zápis definice tohoto signálu je podobný zápisu definice obyčejné metody, s tím rozdílem, že v parametrech této metody se nepíšou názvy parametrů, ale pouze jejich typy. Příklad zápisu je zobrazen ve výpisu kódu 8, kde je vidět část kódu hlavičkového souboru třídy `DataLoader` obsahující definici signálů. Třída, která bude nějaký signál přijímat, musí mít definovanou třídní metodu typu `Qt::SLOT`. Zápis je v tomto případě naprosto stejný jako zápis definice obyčejné metody. Díky tomu se tyto sloty mohou volat i jako obyčejné metody. Příklad zápisu definice je možné vidět ve výpisu kódu 9. Tyto signály a sloty se nakonec musí nějak propojit. To se provede Qt metodou `connect(Object1, signal, Object2, slot)`. První parametr této metody specifikuje objekt, který bude signál vysílat, druhý parametr je specifikace konkrétního signálů tohoto objektu. Třetí parametr říká který objekt bude signál přijímat a čtvrtý parametr definuje slot tohoto objektu, který signál obslouží. Když je potřeba vyslat nějaký signál, provede se to použitím klíčového slova `emit`, následovaným názvem signálu s potřebnými parametry. Po vyslání signálu se spustí všechny sloty, které byly na tento signál napojené.

Tento mechanismus se v programu používá například při předávání dat po přijetí nových dat a bude popsán v kapitole 7.1.2.

```
class DataLoader : public QObject {
    Q_OBJECT
    ...
    signals:
       void loaded(CarData);
       void loaded(vector<CarData>);
    ...
};
```

Výpis 8: Definice signálů v hlavičkovém souboru `DataLoader.h`

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

class MainWindow : public QMainWindow
{
    Q_OBJECT
    ...
    private slots:
      void updateData(vector<CarData> data);
      void updateSingleData(CarData);
      void mousePressed(QMouseEvent *ev);
      void mouseMoved(QMouseEvent *ev);
    ...
}

#endif // MAINWINDOW_H
```

Výpis 9: Definice některých slotů v hlavičkovém souboru `MainWindow.h`

7.1.2 Předávání dat po přijetí nových dat

Po přijetí nových dat se tyto data předávají z instance třídy `DataLoader` instanci třídy `MainWindow`. Předání se provádí pomocí Qt signálů. Použití Qt signálů je vysvětleno v kapitole 7.1.1. Třída `DataLoader` má dva třídní signály: `loaded(CarData)` a `loaded(vector<CarData>)`. První signál se používá pro předání pouze jednoho záznamu a druhý signál se používá pro předání několika záznamů v jednom vectoru. Při načítání dat ze souboru je používán druhý signál. Nejprve se načtou všechna data do vectoru a ten se pomocí tohoto signálu pošle instanci `MainWindow`. Při načítání dat ze sítě se také používá druhý signál, protože v jednom paketu jsou většinou 4 záznamy. Tyto 4 záznamy se uloží do vectoru, který se opět pošle druhým signálem. Změna nastává u načítání dat ze sériové linky. Zde posílání dat probíhá po jednom záznamu, tudíž se používá první signál.

Další předávání dat probíhá při aktualizaci grafů. Zde se však nepředávají data, ale pouze reference na data. Je to z důvodu, aby se data zbytečně nekopírovala a byla na jednom místě. Jak již bylo zmíněno, v konstruktoru třídy `Chart` se v parametrech předá ukazatel na vector s daty. V konstruktorech potomků této třídy se předává navíc další parametr – ukazatel do struktury. Tento ukazatel ukazuje na položku ve struktuře `CarData`, kterou má daný graf zobrazovat. Tento způsob je vynikajícím řešením jak specifikovat, kterou položku bude graf zobrazovat. Šlo by to vyřešit například použitím konstrukce switch nebo několika podmínek, ale tento způsob není praktický, protože by se musely pro každou položku vytvářet nové podmínky. Použitím

ukazatele do struktury se tento způsob stává univerzálním. O tom, která položka se v grafu zobrazí, se rozhodne při inicializaci objektu v konstruktoru. Příklad inicializace jednotlivých typů grafů je uveden ve výpisu kódu 10 a příklad dereference hodnoty z vectoru dat je uveden ve výpisu kódu 11.

```
// MainWindow.h
#define MEMBER_PTR( Type, Element ) ( &( ( Type * ) NULL )->Element )
typedef int16_t CarData::*struct_int_ptr;
typedef uint16_t CarData::*struct_uint_ptr;

// MainWindow.cpp
struct_int_ptr servo = NULL, pwm1 = NULL, pwm2 = NULL;
struct_uint_ptr adc1 = NULL, adc2 = NULL, image = NULL;
( * ( void ** ) &servo ) = MEMBER_PTR( CarData, structControl.servo_pos[ 0 ] );
( * ( void ** ) &pwm1 ) = MEMBER_PTR( CarData, structControl.pwm[ 0 ] );
( * ( void ** ) &pwm2 ) = MEMBER_PTR( CarData, structControl.pwm[ 1 ] );
( * ( void ** ) &adc1 ) = MEMBER_PTR( CarData, structData.adc[ anFB_A ] );
( * ( void ** ) &adc2 ) = MEMBER_PTR( CarData, structData.adc[ anFB_B ] );
( * ( void ** ) &image ) = MEMBER_PTR( CarData, structData.image );

ChartImg("Camera", 128, &data, image));
ChartImg("Filtered Camera", 128, &data, image, &ChartImg::otsu));
ChartInt("Servo", 250, -1000, 1000, QColor(Qt::red), &data, servo));
ChartInt("Left Motor", 150, -1000, 1000, QColor(Qt::blue), &data, pwm1));
ChartInt("Right Motor", 150, -1000, 1000, QColor(Qt::green), &data, pwm2));
ChartUint("Left FB", 150, 0, 4096, QColor(Qt::yellow), &data, adc1));
ChartUint("Right FB", 150, 0, 4096, QColor(Qt::darkMagenta), &data, adc2));
```

Výpis 10: Příklad inicializace objektů grafů

```
// Chart.h
vector<CarData> *data;

// ChartInt.cpp
void ChartInt::renderRow(int y, int dataIndex)
{
    int16_t value = data->at(dataIndex).*ptrToData;
    value /= scaleFactor;

    const int center = width / 2;
```

```
    if (value >= 0)
        for (int x = 0; x < value; x++)
            qimg->setPixelColor(center + x, y, color);
    else
        for (int x = 0; x <= -value; x++)
            qimg->setPixelColor(center - x, y, color);

    qimg->setPixelColor(center, y, QColor(Qt::white));
}
```

Výpis 11: Příklad použití ukazatele na vector s daty s ukazatelem do struktury `CarData`

7.2 Ukládání dat do souboru

Při zaznamenávání dat ze sítě nebo ze sériové linky je ve většině případů nutné si zaznamenaná data někam uložit pro pozdější zobrazení. Běžný způsob, jak ukládat nějaký přijatý záznam je, že se uloží celý záznam. V některých případech ale může být ukládání celého záznamu zbytečné až nežádoucí. Typickým příkladem může být situace, kdy se spustí načítání dat dříve, než auto začne jezdit nebo v opačném případě se nechají data načítat i po projetí cílem. V těchto případech jsou některé části záznamu zbytečné a je zapotřebí uložit pouze jízdu od startu po cíl. Z těchto důvodů bylo nutné implementovat uložení pouze části záznamu.

Jsou tedy dvě možnosti jak záznam uložit: celý záznam nebo pouze část záznamu. Tyto možnosti jsou podrobně popsány v následujících podkapitolách. O tom, zda se bude ukládat celý záznam nebo pouze část záznamu, je rozhodnuto v momentě potvrzení uložení. Pokud je vybrána pouze část záznamu, ukládá se pouze tato část. Pokud není, ukládá se celý záznam. Pro uložení záznamu se používá položka Save to file v menu File v horní části okna. Pro rychlejší přístup je možno použít klávesovou zkratku `CTRL+S`. Otevře se dialog pro výběr cílového adresáře a názvu souboru. Po potvrzení se vybraný úsek dat resp. celý záznam uloží do tohoto souboru a zobrazí se okno informující o úspěchu.

7.2.1 Uložení části záznamu

Pokud je zapotřebí uložit pouze část záznamu, musí se požadovaná část nějak označit. Je možné to provést dvěma způsoby. Oba způsoby jsou velice podobné označování souborů, textu apod. ve většině operačních systémů nebo aplikací. Jedná se tedy o označování pouze myší nebo v kombinaci s klávesou `Shift`.

Při označování pouze myší se požadovaná oblast označuje stisknutím a držením pravého tlačítka a následného přetažení myši na žádané místo. Tento způsob je vhodnější pro označování oblasti, která je v okně aktuálně viditelná. Pokud je zapotřebí označení místa, které je mimo viditelnou oblast, je nutné se posunout pomocí rolovacího kolečka při stálém držení pravého

tlačítka, což je mírně nepohodlné. Z tohoto důvodu je v aplikaci možnost použití klávesy `Shift`. Výběr se provádí označením dvou bodů. Oblast mezi těmito body se označí. Pro vybrání prvního bodu se jednoduše stiskne pravé tlačítko myši nad daným místem. Řádek záznamu se vyznačí. Opakovaným klikáním pravým tlačítkem se mění pozice prvního bodu. Vybrání druhého bodu se provádí stisknutím a podržením klávesy `Shift` a kliknutím pravým tlačítkem myši nad požadovaným místem. Vyznačí se celá oblast od prvního do druhého bodu.

Důvod, proč se pro označování oblasti nepoužívá levé tlačítko, jak je tomu např. při označování souborů, je jednoduchý. Levé tlačítko se v aplikaci používá pro výběr jednoho řádku záznamu, pro který se mají zobrazit podrobnější informace. Kromě toho má ale ještě jednu důležitou funkci, a to rušení označené oblasti. Pokud je označená nějaká oblast pro uložení, případně vybraný pouze první bod oblasti, tak po stisknutí levého tlačítka se tato označení zruší.

7.2.2 Uložení celého záznamu

V případě, že je požadováno uložení celého záznamu, musí se nejdříve zrušit jakékoliv označení oblasti pro uložení. Provede se to vybráním jakéhokoliv řádku záznamu levým tlačítkem, čímž se jakékoliv označení pro uložení zruší. Poté je možno celý záznam uložit stejným způsobem, který byl popsán v první části kapitoly 7.2.

8 Grafické uživatelské rozhraní

V začátcích vývoje aplikace jsem pro tvrobu grafického uživatelského rozhraní používal nástroj Qt Designer. Tento nástroj jsem používal především z důvodu, že nabízel pohodlnou tvorbu rozhraní, rozvržení jednotlivých grafických prvků a rychlou změnu parametrů těchto prvků. Veškeré změny v návrhu byly ihned vidět, nebylo nutné aplikaci kompilovat a spouštět. Tento přístup byl dostatečný do doby, než bylo potřeba programově přidávat další grafické prvky za běhu programu a poté s nimi manipulovat. Vzhledem k tomu, že některé grafické prvky byly definovány v Qt Designeru a některé později až za běhu programu, tak byla tvorba GUI rozdělena na několika místech a stala se tak nepřehlednou. Z tohoto důvodu jsem se rozhodl veškerou tvorbu GUI provádět programově v konstruktoru třídy `MainWindow`. Díky tomu jsem si mohl deklaraci a inicializaci grafických prvků logicky seskupit, oddělit odřádkováním a okomentovat. Příklad tvorby některých prvků GUI v konstruktoru třídy `MainWindow` je uvedena ve výpisu kódu 12.

```
// Menu File creation
QMenu *menuFile = new QMenu("File");
QAction *actionSaveToFile = new QAction("Save to file");
QAction *actionExit = new QAction("Exit");

actionSaveToFile->setShortcut(QKeySequence("Ctrl+S"));
actionExit->setShortcut(QKeySequence("Ctrl+Q"));

menuFile->addAction(actionSaveToFile);
menuFile->addSeparator();
menuFile->addAction(actionExit);

ui->menuBar->addMenu(menuFile);

// Creation of header containing Timestamp information
QFrame *informationHeader = new QFrame;
informationHeader->setLayout(new QHBoxLayout);

timestampLabel = new QLabel("Timestamp:");
informationHeader->layout()->addWidget(timestampLabel);

ui->centralWidget->layout()->addWidget(informationHeader);
```

Výpis 12: Příklad tvorby prvků GUI

Výsledné hlavní okno aplikace s načtenými a zobrazenými daty je zobrazeno na obrázku 4. V následujících podkapitolách budou jednotlivé prvky grafického uživatelského rozhraní popsány.

Obrázek 4: Okno programu s načtenými daty ze souboru

8.1 Hlavní okno

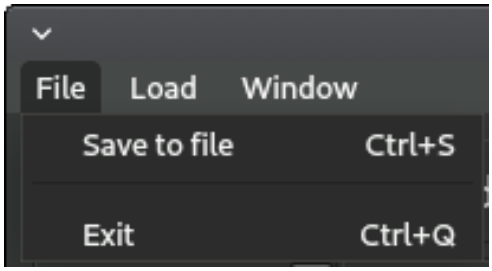
Hlavní okno obsahuje menu, dvě sekce s informacemi o aktuálně vybraném záznamu a oblast s grafy.

8.1.1 Menu

Menu obsahuje tři podmenu – File, Load a Window.

Menu File, zobrazeno na obrázku 5, obsahuje akce:

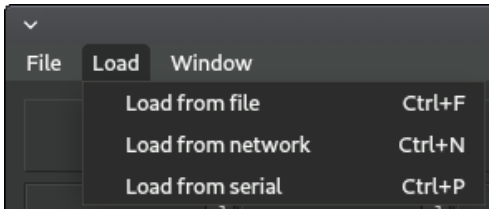
- Save to file - zobrazení dialogu umožňující vybrat název souboru a adresář, do kterého se poté uloží soubor se záznamem
- Exit - zavření aktuálního okna. Pokud je v kontextu aplikace otevřeno pouze jedno okno, ukončí se celá aplikace. Pokud je otevřeno oken více, tak tato akce zavře pouze aktuální okno, ostatní okna a program dále poběží.



Obrázek 5: Menu File

Menu Load, zobrazeno na obrázku 6, obsahuje akce:

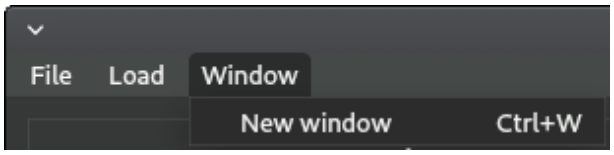
- Load from file - zobrazení dialogu umožnující vybrat datový soubor obsahující data
- Load from network - zobrazení dialogu umožňující zvolit síťový port. Jako výchozí port je nastaven port 12345.
- Load from serial - zobrazení dialogu umožňující vepsat cestu k portu. Jako výchozí port je nastaven port `/dev/ttyS0`.



Obrázek 6: Menu Load

Po vybrání kterékoliv akce a vyplnění potřebných údajů aplikace zkontroluje zda byl vstup validní a zda je možné provést vybranou akci. Pokud ne, zobrazí se chybové okno informující, která chyba nastala. Pokud ano, provede se akce, např. začnou se načítat data ze sériové linky.

Menu Window, které je možno vidět na obrázku 7, obsahuje pouze jednu akci a to New window, která otevře nové okno aplikace umožňující zobrazovat jiná data, např. z jiného souboru, z jiného síťového portu, atd.



Obrázek 7: Menu Window

8.2 Informační hlavička

Informační hlavička slouží k poskytnutí informací o aktuálně vybraném záznamu, které je potřeba zobrazit, ale není nutné pro ně vykreslovat graf, například z důvodu, že by takový graf neměl smysl.

Ve výchozím stavu tato hlavička zobrazuje tři informace:

- Index - číslo určující pozici záznamu v datech
- Timestamp - časová známka záznamu
- Battery - napětí na baterii

8.3 Statické záznamy

Tato oblast obsahuje statické záznamy pro jednotlivé grafy. Statické záznamy jsou umístěny v horizontálním Qt layoutu.

Ve výchozím stavu tento layout obsahuje následující statické záznamy:

- Záznam kamery - typ `StaticRecordImg`
- Záznam filtrované kamery - typ `StaticRecordImg`
- Servo - typ `StaticRecordInt`
- Levý motor - typ `StaticRecordInt`
- Pravý motor - typ `StaticRecordInt`
- Levou zpětnou vazbu - typ `StaticRecordInt`
- Pravou zpětnou vazbu - typ `StaticRecordInt`

8.4 Grafy

Poslední oblast v okně se stará o vykreslení jednotlivých grafů. Tato oblast opět obsahuje horizontální Qt layout, do kterého se objekty grafů naskládají při vytvoření hlavního okna.

Ve výchozím stavu tento layout obsahuje následující grafy:

- Záznam kamery - typ `ChartImg`
- Záznam filtrované kamery - typ `ChartImg`
- Servo - typ `ChartInt`
- Levý motor - typ `ChartInt`
- Pravý motor - typ `ChartInt`

- Levý feedback - typ `ChartUint`
- Pravý feedback - typ `ChartUint`

9 Problémy při implementaci a jejich řešení

Při implementaci nastalo několik problémů, které bylo potřeba vyřešit. Tyto problémy jsou popsány v následujících podkapitolách.

9.1 Výpočetní náročnost

V první verzi programu se obrázky grafů po každé aktualizaci dat celé překreslovaly. Tento způsob byl použit, protože byl nejjednodušší na implementaci a také proto, že jako první se implementovalo načítání dat ze souboru, kde se nejprve načetla všechna data do vectoru a poté se jednorázově zobrazila. Avšak při implementaci příjmu dat přes UDP protokol nastal problém. Když se obrázky překreslovaly ihned po přijetí nových dat, tak to vyžadovalo velký počet nových výpočtů – musel se znovu překreslovat stále větší a větší obrázek, což po chvíli přijímání způsobilo zamrznutí celé aplikace. Tento problém byl dočasně vyřešen tak, že se obrázky překreslovaly každých 10 záznamů. Toto řešení však nebylo dostatečné a bylo zapotřebí vymyslet jiný způsob. Řešení bylo ihned zřejmé. Bylo nutné zachovat už vykreslený obrázek a následně do něj pouze dokreslovat nová data.

První možnost, jak tohoto dosáhnout, je vytvořit obrázek o výšce odpovídající aktuálnímu počtu načtených dat. Při načtení nových dat by se musel obrázek zvětšit a dokreslit do něj. Tento způsob s sebou přinesl problém. Již vytvořený obrázek není možné jednoduše zvětšit. Je potřeba vytvořit nový obrázek o velikosti starého + velikosti nových dat, poté starý obrázek překopírovat do nového, dokreslit nová data a objekt starého obrázku smazat. Tento způsob je ale také náročný na výkon. Při počtu záznamů v UDP paketu 4 a objemu dat 10000 záznamů by se obrázek musel zvětšovat 2500 krát.

Poslední vyzkoušenou možností je následující postup. Nejprve se vytvoří obrázek o počáteční výšce definované konstantou `IMAGE_RESIZE_STEP`, která je v základu nastavena na 1000. Do něj se postupně vykreslují data. Obrázek se musí zobrazit ořezaný, aby nebyl vidět prázdný prostor určený pro nová data. Po využití dostupného prostoru v obrázku se vytvoří nový obrázek, zvětšený o dříve zmíněnou konstantu, zkopíruje se do něj obsah starého obrázku a objekt starého obrázku se smaže. Při tomto řešení je sice nutné zobrazovat ořezaný obrázek, ale odpadá nutnost neustálého vytváření nového objektu obrázku. Nyní se při objemu dat použitém v předchozím příkladě obrázek zvětší pouze 10 krát.

9.2 Zpožděné pakety

V případě příjmu dat přes UDP protokol může nastat situace, že některé pakety mohou přijít opožděně, tudíž bez správného algoritmu budou ve výsledku neuspořádané. Tento problém se dá vyřešit zpožděným vykreslováním, kdy se na chybějící paket čeká určitou dobu specifikovanou buď časem nebo počtem záznamů. Já jsem se vydal cestou počtu záznamů.

Zde jsem vymyslel dva algoritmy, které podrobně popíšu. Oba algoritmy používají buffer o velikosti `N * STRUCTS_IN_PACKET`, kde `N` je ve výchozím stavu 50 a `STRUCTS_IN_PACKET` 4. Aktualizace grafů se provádí vysláním signálu `loaded(vector<CarData>)`, kde se v prvním parametru předá vector s daty. Dále je zde používána proměnná `prevTimestamp`, do které se vždy vloží časová známka posledního záznamu, který byl poslán signálem grafům. Tato proměnná se nadále využívá pro zjištění posloupnosti záznamů.

9.2.1 První algoritmus

Po přijetí prvního paketu se data z něj ihned pošlou grafům a do proměnné `prevTimestamp` se vloží hodnota časové známky posledního záznamu v paketu. Po přijetí dalších paketů už se musí kontrolovat, zda se jedná o souvislá data. Provede se kontrola, zda je časová známka prvního záznamu v paketu o jedno číslo větší než hodnota proměnné `prevTimestamp` a zda se v bufferu nenachází žádné záznamy. V případě pozitivního výsledku obou testů se data opět pošlou grafům a nastaví se proměnná `prevTimestamp`. V opačném případě se projde buffer od začátku do konce a hledá se místo, kde daná data z paketu patří. Po nalezení se data na dané místo vloží. Jeden z posledních testů je test počtu prvků v bufferu a posloupnosti dat. Následně se provedou dva testy a pokud výsledek alespoň jednoho z nich bude pozitivní, tak se zavolá metoda pro odebrání dat z bufferu a aktualizování grafu – `popFromBuffer()`. První test je kontrola, zda je počet prvků v bufferu větší než `N * STRUCTS_IN_PACKET`. Druhým testem se zkontroluje, zda je časová známka prvního záznamu v bufferu o jedno číslo větší než proměnná `prevTimestamp`. Pozitivní výsledek tohoto testu znamená, že nám konečně přišla očekávaná data.

Metoda `popFromBuffer()` nejprve zjistí, zda je první prvek v bufferu očekávaným prvkem s časovou známkou o jedno číslo větší než `prevTimestamp`. Pokud ne, tak to znamená, že metoda se zavolala z důvodu překročení velikosti bufferu, na chybějící záznam už se čekat nebude a musí se tedy označit výpadek. To se provede vložením prázdného záznamu do dat. Poté má tato metoda za úkol projít buffer a najít souvislou sekvenci dat. Po nalezení nebo po projití celého bufferu se sekvence souvislých dat, včetně možného prázdného záznamu, pošle grafům pro zobrazení a smaže se z bufferu.

9.2.2 Druhý algoritmus

Druhý algoritmus se liší v tom, jak nakládá s přijatými pakety a jak je maže z bufferu. Na rozdíl od prvního algoritmu se přijatá data střádají v bufferu i v případě, že se jedná o souvislou sekvenci dat. První přijatý paket se ihned uloží na první místo do bufferu a proměnná `prevTimestamp` se nastaví na číslo o jedno menší než je první záznam v paketu. Nastavení této proměnné se dělá z důvodu kontroly, zda se jedná o první přijatý paket. Před přijetím prvního paketu je hodnota této proměnné 0. Další přijatá data si už musí najít své místo v bufferu. Pokud je časová známka prvního záznamu v paketu větší než časová známka posledního záznamu v bufferu, tak se přijatá data uloží na konec bufferu. Pokud je menší, tak se prochází buffer od

začátku do konce a hledá se záznam, který má časovou známku větší a po nalezení se přijatá data vloží před tento nalezený záznam. Na konci se provede test, zda počet prvků v bufferu nepřekročil danou velikost. Pokud ano, tak se zavolá metoda `popFromBuffer()`. Tato metoda stejně jako u prvního algoritmu ověří pomocí proměnné `prevTimestamp`, zda se jedná o souvislou sekvenci a pokud ne, tak na začátek dat vloží prázdný záznam. Zde nastává velká změna oproti prvnímu algoritmu. Grafům se neposílá celá souvislá sekvence, ale pouze data odpovídající prvnímu paketu – v našem případě tedy 4 záznamy, případně 5, pokud je nutný i prázdný záznam. Tyto data se poté smažou z bufferu.

9.2.3 Závěr

Druhý přístup je uživatelsky mnohem přívětivější než první, protože zobrazování dat není trhané. Představme si, že při přijímání paketů se jeden paket ztratí. Další souvislé pakety se střádají v bufferu dokud se buffer nepřeplní. Mezitím se žádná data nezobrazují. Po přeplnění bufferu se zjistí, že celý buffer obsahuje souvislou sekvenci záznamů a grafům se tedy pošle celá tato sekvence. Uživatel tedy vidí jak se postupně zobrazují data, poté se zobrazování zastaví a po přeplnění bufferu se zobrazí velké množství nových dat najednou. V případě velkého počtu výpadků je trhavost zobrazování ještě horší, protože střídání mezi zobrazováním dat a čekáním je nepravidelná.

U druhého algoritmu sice nevidíme nejaktuálnější data, protože čekáme než se naplní buffer, ale na druhou stranu díky postupnému vyprazdňování bufferu po jednom paketu je zobrazování dat pěkně plynulé.

Pokud by bylo jisté, že výpadků bude málo, tak by bylo výhodnější použít první algoritmus. Při použití UDP protokolu to však nelze zajistit, čili je mnohem výhodnější použít druhý algoritmus.

10 Testování

Při vývoji aplikace jsem měl k dispozici soubory s testovacími daty. Těmito soubory jsem testoval funkčnost čtení dat ze souboru. Poté přišlo na řadu testování příjmu dat přes UDP protokol. Jelikož jsem neměl k dispozici model auta na testování, tak jsem používal jednoduchý program naprogramovaný rovněž v programovacím jazyce C++, který čte data ze souboru a posílá je přes UDP protokol do sítě. Tento přístup byl velmi užitečný a příjem dat jsem dostatečně otestoval a vyladil. Poté nastal čas na otestování tohoto způsobu příjmu dat s reálným modelem auta.

10.1 Testování s modelem auta

První test probíhal na oválné dráze. Při testování se zjistilo, že některá data chodí opožděně a bylo tedy nutné naimplementovat řešení tohoto problému. Řešení bylo popsáno v kapitole 9.2. Také se přišlo na několik funkčních věcí, které by mohly být doimplementovány, především pozastavení a zastavení příjmu nebo vylepšení ukládání přijatých dat do souboru. Vylepšení spočívá v možnosti označit pouze úsek dat, který chceme následně uložit. Popsání tohoto vylepšení proběhlo v kapitole 7.2.

Druhý test proběhl na soutěži NXP Cup, kde v rámci trénování a testování na závěrečnou soutěž byly postaveny dvě dráhy. První dráha měla tvar osmičky s křižovatkou a je ukázána na obrázku 8. Druhá dráha byla rozsáhlejší s několika obtížnými pasážemi a je možno ji vidět na obrázku 9.

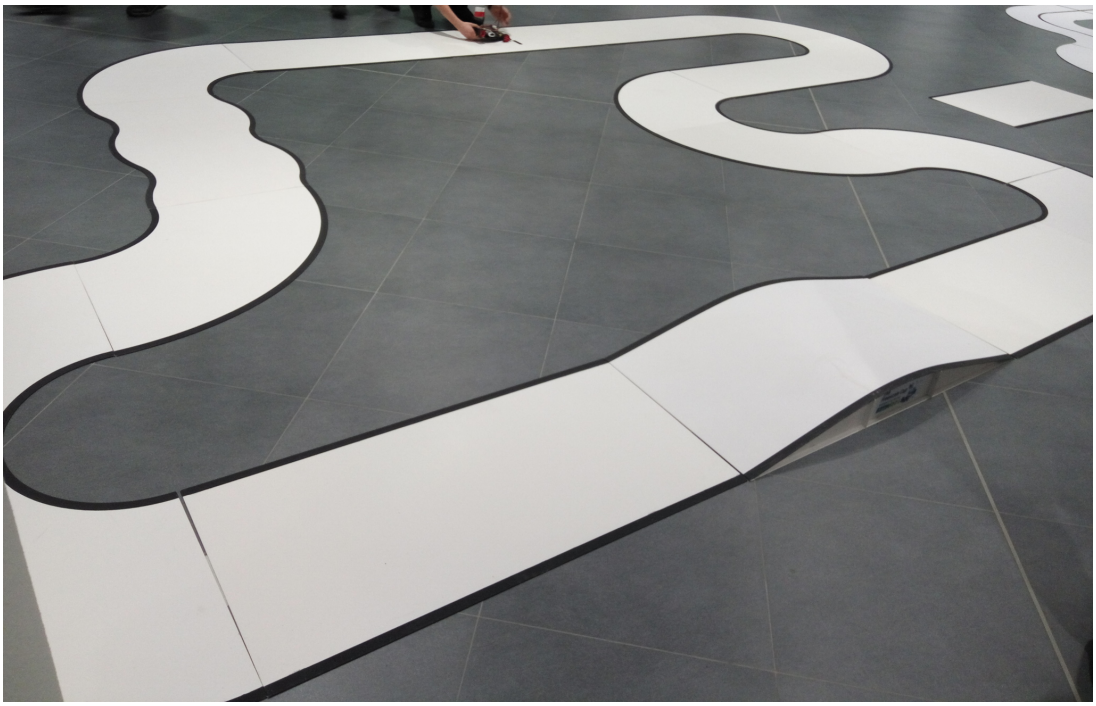
10.2 Testování výpadků paketů

Při návrhu algoritmů popsaných v kapitole 9.2 bylo zapotřebí tyto algoritmy otestovat za účelem zjištění, který algoritmus je efektivnější. V rámci tohoto testování jsem se zaměřil především na ztrátovost paketů. Abych mohl provést objektivní vyhodnocení, musel jsem použít stejná testovací data pro oba algoritmy. To by ovšem nebylo možné, pokud bych testování prováděl při reálném příjmu dat přes UDP, protože by nikdy nenastaly stejné podmínky pro každé testování. Z tohoto důvodu jsem opět použil simulaci odesílání dat pomocí jednoduchého programu, který jsem již popsal na začátku kapitoly 10.

V obou algoritmech se pro načítání dat používá buffer, ale každý algoritmus s tím bufferem manipuluje jiným způsobem. U tohoto bufferu jsem střídavě měnil jeho velikost. Tímto jsem hledal hraniční hodnotu, při které je velikost bufferu co nejmenší, ale zároveň se nezačnou zbytečně zahazovat některé pakety. K velkému překvapení jsem zjistil, že touto hraniční hodnotou je hodnota tři. Když velikost bufferu odpovídala velikosti třech paketů, tak oba algoritmy nezahodily jediný paket. Při velikosti bufferu odpovídající dvěma paketům se již pakety zahazovaly a počet chybějících snímků stoupl na více než dvojnásobek.

Zjištěná hraniční hodnota ovšem platí pouze pro testování pomocí simulace odesílání dat přes síť. Při reálném používání mohou mít pakety mnohem větší zpoždění, tudíž jako velikost

Obrázek 8: První dráha



Obrázek 9: Druhá dráha

bufferu je možno zvolit velikost odpovídající 25 paketům, případně i větší. Při takovéto hodnotě bude uživatelský komfort stále dobrý a zpoždění zobrazování dat bude krátké.

Při testování jsem rovněž zjistil, že oba algoritmy mají ztrátovost naprosto stejnou. Z tohoto důvodu je mnohem výhodnější používat druhý algoritmus, který byl popsán v kapitole 9.2.2, protože zobrazování dat je plynulé.

11 Závěr

Cílem bakalářské práce bylo vytvořit aplikaci s grafickým uživatelským rozhraním pro záznam a zobrazení zaznamenaných jízdních parametrů modelu auta. Tento cíl byl úspěšně splněn.

Aplikace umožňuje načítat zaznamenaná data ze souboru, umí načítat nová data ze sítě nebo ze sériové linky a tyto data, případně část dat, uložit do souboru pro pozdější zobrazení. Aplikace je stabilní, poněvadž je ošetřena proti různým chybovým stavům. Přidávání dalších zobrazovaných údajů se provádí přidáním pouhých čtyř řádků kódu v jednom místě, tudíž je přidávání pohodlné a především jednoduché. Ovládání aplikace je rovněž snadné, uživatelské rozhraní je intuitivní, funkce ovládacích prvků jsou dle jejich názvů zřejmé. Naučením a použitím klávesových zkratek, které ve většině případů korespondují s těmi užívanými v operačním systému, se interakce s aplikací ještě více urychlí a usnadní. Aplikace je přehledná, rozmístění ovládacích prvků je logické. Při spuštění aplikace se velikost okna volí tak, aby byly vidět všechny ovládací a zobrazovací prvky. Tuto velikost je poté pro větší přehlednost při více otevřených oknech možno zmenšit, případně je možné některé zobrazované údaje schovat.

Do budoucna by se aplikace dala ještě více uživatelsky zpřístupnit, například přidáním grafických ovládacích prvků umožňujících změnu některých proměnných nebo konfiguraci zobrazovaných údajů tak, aby bylo možné tyto parametry měnit během běhu aplikace bez nutnosti jakékoliv změny v kódu programu.

Literatura

[1] Qt Documentation [online]. The Qt Company, c2017 [cit. 2018-04-25]. Dostupné z: http://doc.qt.io/

[2] GTK+ Documentation [online]. The GTK+ Team, c2007-2017 [cit. 2018-04-25]. Dostupné z: https://www.gtk.org/documentation.php

[3] OpenCV library [online]. OpenCV team, c2018 [cit. 2018-04-27]. Dostupné z: https://opencv.org/

[4] C++ Reference [online]. cplusplus.com, c2000-2017 [cit. 2018-04-25]. Dostupné z: http://www.cplusplus.com/reference/

[5] SOMMERVILLE, Ian. Softwarové inženýrství. Brno: Computer Press, 2013. ISBN 978-80-251-3826-7.

[6] FOWLER, Martin. Destilované UML. Praha: Grada, 2009. Knihovna programátora (Grada). ISBN 978-80-247-2062-3.

A Hlavičkový soubor `tfc.h`

```
/**
@file tfc.h
*/

#ifndef __TFC_H
#define __TFC_H

#include <stdint.h>

#define TFC_VERSION "20171213"

#define __TFC_EMBEDED__ 1

/** Resolution of the camera. */
#define TFC_CAMERA_LINE_LENGTH  128

/** Specifies the integer range for PWM, as in <-::TFC_PWM_MINMAX, ::
    TFC_PWM_MINMAX>, which is used in this application's methods
    TFC::getMotorPWM_i() and TFC::setMotorPWM_i().
*/
#define TFC_PWM_MINMAX        1000

/** Specifies the integer range for servo, as in <-::TFC_SERVO_MINMAX, ::
    TFC_SERVO_MINMAX>, which is used in this application's methods
    TFC::getServo_i() and TFC::setServo_i().
*/
#define TFC_SERVO_MINMAX      1000

/** Maximum value of analog sample obtained from ADC by the method TFC::ReadADC
    (). */
#define TFC_ADC_MAXVAL        0x0FFF

/** Specifies the integer ranges for analog values, as in <-::TFC_ANDATA_MINMAX
    , ::TFC_ANDATA_MINMAX> or <0, ::TFC_ANDATA_MINMAX>, which are used in this
    application's methods
    TFC::ReadPot_i(), TFC::ReadFB_i() and TFC::ReadBatteryVoltage_i().
*/
```

```
#define TFC_ANDATA_MINMAX     1000

/** Default center position of servos - pulse width in microseconds. */
#define TFC_SERVO_DEFAULT_CENTER 1500

/** Specifies the default offset from ::TFC_SERVO_DEFAULT_CENTER, which
    corresponds to fully turned wheels to either side. */
#define TFC_SERVO_DEFAULT_MAX_LR 200

/** Specifies the maximal offset from the servos' center, which can be used in
    calibration by method TFC::setServoCalibration(). */
#define TFC_SERVO_MAX_LR      400

/** Specifies the default integer range for PWM, as in <-::TFC_PWM_DEFAULT_MAX,
     ::TFC_PWM_DEFAULT_MAX>, which can be changed in the method
    TFC::setPWMMax() up to <-::TFC_PWM_MINMAX, ::TFC_PWM_MINMAX>.
*/
#define TFC_PWM_DEFAULT_MAX     200

/** Maximal allowed -/+ PWM duty cycle for the underlying HW. */
#define HW_TFC_PWM_MAX        500

/** Command used in a packet for sending/receiving data. */
#define CMD_DATA           1

/** Command used in a packet for calibration. */
#define CMD_SETTING           2

/** Command used in a packet for controlling features of the board or receiving
     data from them. */
#define CMD_CONTROL           3

/** Packet header. */
#define STX                0x2
/** Packet footer. */
#define ETX                0x3

/** @brief Order of analog data in an array. These values are used as a
    parameter for a method TFC::ReadADC(). */
```

```
enum tfc_andata_chnl_enum
{
    anPOT_1,  ///< Potentiometer 1.
    anPOT_2,  ///< Potentiometer 2.
    anFB_A,   ///< Feedback A from the H-Bridge.
    anFB_B,   ///< Feedback B from the H-Bridge.
    anBAT,    ///< Battery voltage.
    anLast    ///< Last member used as a size of an array.
};


/**
@brief Empty data packet

This packet contains only necessary parameters, no data. \n
When using variation of this packet to send/receive data, the data should be
    put in a place of ::data. \n
There can be some of the data structure (::tfc_data_s, ::tfc_setting_s or ::
    tfc_control_s) in one packet.

@code
  Protocol specification:
    length = 1 + 1 + 2 + N + 1
    bytes   1      2       1         N        1
    types uint_8 uint16_6 uint8_t  N*uint8_t uint8_t
   purpose STX    length  CMD_xx    struct     ETX
@endcode
*/
#pragma pack(push,1)
struct tfc_protocol_empty_s
{
    uint8_t stx; ///< Each packet must start with the ::STX byte.
    uint16_t length;///< Length of the whole packet.
    uint8_t cmd; ///< Type of command, can be either ::CMD_DATA, ::CMD_SETTING
        or ::CMD_CONTROL.
    char data[0]; ///< Placeholder for a data.
    uint8_t etx; ///< Each packet must end with the ::ETX byte.
};
```

```
/**
@brief Structure containing analog data and data from camera.

This data structure contains data corresponding to a certain timestamp. It is s
    usually sent in packet (::tfc_protocol_empty_s) with ::CMD_DATA. \n
*/
struct tfc_data_s
{
    uint32_t  timestamp;              ///< Number of the sample.
    uint16_t  adc[ anLast ];          ///< All analog data specified in ::
       tfc_andata_chnl_enum.
    uint8_t   dip_sw;                 ///< Values of DIP switches.
    uint8_t   push_sw;              ///< Values of push buttons.
    uint16_t  image[ TFC_CAMERA_LINE_LENGTH ];///< One line from the camera.
    uint32_t  _padding;               ///< Padding bytes.
};

/**
@brief Structure for calibration of the individual features.

This data structure is usually sent in packet (::tfc_protocol_empty_s) with ::
    CMD_SETTING in the beginning of an application. \n
*/
struct tfc_setting_s
{
    uint16_t  servo_center[ 2 ];      ///< Center of the servos.
    uint16_t  servo_max_lr[ 2 ];      ///< Offset from the center of the servos
        corresponding to their maximal rotation.
    uint16_t  pwm_max;           ///< Maximal PWM for motors.
    uint16_t  _padding;             ///< Padding bytes.
};

/**
@brief Structure for controlling individual features or receiving their data.

This data structure is usually sent in packet (::tfc_protocol_empty_s) with ::
    CMD_CONTROL. \n
*/
struct tfc_control_s
```

```
{
    uint8_t   leds;              ///< Values of the 4 LEDs in the form of a low
        nibble.
    uint8_t   pwm_onoff;           ///< On/off value of the motors {0,1}.
    uint8_t   servo_onoff;         ///< On/off value of the servos {0,1}.
    uint8_t   _padding1;           ///< Padding byte.
    int16_t   pwm[ 2 ];            ///< PWM values of the motors <-::
       TFC_PWM_MINMAX, ::TFC_PWM_MINMAX>.
    int16_t   servo_pos[ 2 ];         ///< Positions of the servos <-::
       TFC_SERVO_MAX_LR, ::TFC_SERVO_MAX_LR>.
};
#pragma pack(pop)

#endif // __TFC_H
```

Výpis 13: Výpis hlavičkového souboru `tfc.h`

B CD

K této bakalářské práci je přiloženo datové CD, které obsahuje následující soubory:

- Text - Text bakalářské práce
- Aplikace - Zdrojové soubory aplikace
- Datové soubory - Ukázkové datové soubory, které byly použity pro testování